User Experiences with the Aonix ObjectAda RAVEN
Ravenscar Profile Implementation

David Brach, P.Eng

CMC Electronics Inc.
Ottawa, Canada

E-Mail : { Dave.Brach@cmcelectronics.ca }

Abstract

CMC Electronics Inc. has gained experience using the Ravenscar Profile by developing avionics software using the Aonix ObjectAda RAVEN toolset. The Aonix toolset was selected by CMC Electronics for several reasons including familiarity with the product family, the well-known IDE, the small kernel footprint, the claimed Ravenscar Profile compatibility (at the time of evaluation the only claimant) and the certifiability option. The Aonix toolset met all our criteria for an avionics software development toolset. From our development experience certain conclusions with respect to the Ravenscar Profile and Aonix toolset have been reached, which will be expressed herein.

1 Introduction

CMC Electronics Inc. develops avionics software for both military and civilian markets. It continues to develop systems that require varying levels of certification. Since each development project is unique we can only talk in terms of past achievements in this domain. CMC has received Transport Canada certification with respect to DO-178A level 3 software on one project, FAA certification on another and has a wealth of experience developing mission critical DO-178B level C software with certain components developed to DO-178B level A safety critical. The Ravenscar Profile, as an industry standard, is of interest as it provides a level of understanding with respect to the capabilities and limitations of the developed software. As such, the Avionics Division at CMC Electronics has chosen the ObjectAda RAVEN toolset from Aonix as the standard Ada95 toolset and is currently using this toolset to develop mission critical software. Respecting the reality that customers will mandate specific toolsets on separate mission critical software development projects, we are working with the Ravenscar Profile using another compiler in order to gain the advantages intrinsic with the principles of Ravenscar. We are proponents of the philosophy laid out in the profile, although we have issues with parts of the profile and the implementation of it.

The Ravenscar Profile in our estimation is essentially a definition of a restricted tasking profile for use in high-integrity real-time systems [8]. The Ravenscar Profile name somewhat obscures the fact that it was developed strictly as a means of quantifying a subset of the tasking components of the language acceptable for the development of real-time software. The publication of the Ravenscar Profile has generated a significant interest in many of the real time software disciplines, which has resulted in the profile being rightly or wrongly interpreted as a de facto standard [1]. This has resulted in misleading use of the term Ravenscar. The Ravenscar Profile is claimed to be capable of being used as a standard for Ada software development in the area of safety critical systems [3]. This image has created critics of the Profile, who have found holes in its definition, some warranted others misplaced. What is becoming apparent is that safety critical software development requires further profiling with respect to areas not covered by the Ravenscar Profile.

These issues, along with our experiences developing Avionics software using the Aonix ObjectAda RAVEN toolset will be discussed in the paper.

2 Aonix Ravenscar

As initially stated, the Aonix product offering was selected for use in developing Avionics software in part due to the Ravenscar pedigree. The implications of attaching the name RAVEN or Ravenscar to a product are that customers will make purchases based on certain assumptions. In this case the assumption was that the provisions within the Profile would be met and those not mentioned would not be restricted. The following is a short analysis of our conclusions with respect to our initial assumptions.

2.1 The Ravenscar Profile Restriction Set

The Ravenscar Profile identified 9 existing and 11 new restrictions to be used with the pragma Restrictions statement in order to restrict the Ada language when developing a system that would be known as Ravenscar compliant. These restrictions are:

Existing Restrictions	**New Restrictions**
No_Task_Hierarchy	Simple_Barrier_Variables
No_Abort_Statements	Max_Entry_Queue_Depth => 1
No_Task_Allocators	No_Calendar
No_Dynamic_Priorities	No_Relative_Delay
No_Asynchronous_Control	No_Protected_Type_Allocators
Max_Task_Entries => 0	No_Local_Protected_Objects
Max_Protected_Entries => 1	No_Requeue
Max_Asynchronous_Select_Nesting => 0	No_Select_Statements
Max_Tasks => N	No_Task_Attributes
	No_Task_Termination
	No_Dynamic_Interrupt_Priorities

Table 1 - Ravenscar Profile Restriction Set

2.2 The Aonix ObjectAda Implementation

The Aonix ObjectAda RAVEN product is comprised of a superset of the Ravenscar restrictions. Using the Aonix terminology, the restrictions are divided into the following groupings; hard restrictions which are not user modifiable, soft restrictions which can be optionally included by the user, and unusable restrictions which are not excluded by the Ravenscar Profile but which are violated by the Aonix runtime and thus cannot be used.

The following is a list of the Aonix hard restrictions:

Ravenscar Restrictions	**Aonix Additional Restrictions**
Max_Asynchronous_Select_Nesting => 0	No_Enumeration_Maps
Max_Entry_Queue_Depth => 1	No_Exception_Handlers
Max_Protected_Entries => 1	No_IO
Max_Task_Entries => 0	No_Nested_Finalization
No_Abort_Statements	No_Standard_Storage_Pools
No_Asynchronous_Control	No_Streams
No_Calendar	No_Wide_Characters
No_Dynamic_Priorities	
No_Local_Protected_Objects	
No_Protected_Type_Allocators	
No_Relative_Delay	
No_Requeue	
No_Select_Statements	
No_Task_Allocators	
No_Task_Attributes	
No_Task_Hierarchy	
No_Task_Termination	

Table 2 - Aonix Hard Restriction Set

Included with the above list of hard restrictions, Aonix has also included the policies FIFO_Within_Priorities and Ceiling_Locking, the setting of which are not modifiable.

The tool allows for the inclusion of many other restrictions as soft restrictions. It is using this mechanism that the user would enable the restriction for Simple_Barrier_Variables.

The unusable restrictions can be found in [2], and are repeated here:

No_Exceptions, No_Unchecked_Conversion, No_Access_Subprograms, No_Unchecked_Access and No_Reentrancy.

2.3 Synopsis

Thus we can see that our initial assumption was not quite correct, that although the Aonix ObjectAda RAVEN product claims to be Ravenscar Profile compliant, it has not implemented all of the 20 defined restrictions within it's hard restrictions. It has placed one restriction in it's group of soft restrictions, namely the Simple_Barrier_Variables restriction, which concerns us because developers are more than likely to miss setting this, inadvertently violating the Profile by omission. Our other concerns can be summarized as follows:

1. A reluctance to be forced into using a specified solution, specifically with respect to exception handling, which the Ravenscar Profile does not discuss;
2. Clarification of why the additional restrictions were imposed, especially with respect to enumeration maps and 'Image and 'Value;
3. The Ravenscar Profile's silence on the exception handling issue has resulted in vendors implementing their own schemes in this area. The fact that we are satisfied with the Aonix implementation is irrelevant and the profile should clarify its intentions in this area.
4. The Aonix ObjectAda RAVEN implements the Ceiling_Locking policy, which is not discussed in the Ravenscar Profile. In rationalizing this we determined that since there is a requirement for bounded blocking and the use of protected objects that the only policy suitable for this is Ceiling_Locking. We think that the Ravenscar Profile should be clearer in this regard.

Simply put, the Aonix ObjectAda RAVEN product meets our needs with respect to a software development system moving in the direction of Ravenscar compliance. We are satisfied with both the Ravenscar Profile and the RAVEN product. Our concerns stem from the fact that vendors are creating supersets of restrictions in the Ravenscar Profile and labelling them as Ravenscar Profile compliant. This is misleading and leads to portability concerns. It is our contention that all vendors should strictly adhere to the implementation of the Ravenscar Profile in order to claim compliance, and that all vendor specific additions should be implemented via the Aonix-like soft restrictions.

3 Ada95 Ravenscar Problems

In general terms our transition to designing using the principles outlined within the Ravenscar Profile has been successful. We did however, experience some growing pains in learning how to best use protected objects, exception handling and class wide type programming.

3.1 Protected Objects

The Ravenscar Profile eliminated the concept of the rendezvous from Ada tasking. Tasks now communicate via shared data encapsulated in or synchronised using the new construct known as the protected object. Our two projects took two separate approaches with respect to using protected objects. One took a data centric approach and used protected objects to encapsulate and protect the shared data while the other project also used the protected objects to synchronise processing. It was found that when using the protected objects to contain the data within them, that the performance of the system was affected by the size of the data being protected. That is, in order to access the data the entire structure was being read and then written, thus the efficiency of the system was dependent on the amount of data being passed through the protected object's interfaces. In order to avoid this overhead, but maintain the data protection provided by the construct, the only alternative thought of was to break the data object up into smaller protected objects. This however is not always possible or desirable. Also, the alternative of simply introducing more procedures to the singe large protected object to allow access to it's smaller components is not practical, as the issue of protected access and multiple writers becomes a design problem.

Our experience with protected objects when trying to implement shared resource control type scenarios has been that even simple paradigms and design patterns involve the use of many protected objects in order to achieve the proper access control of multiple tasks sharing the same resource. The non-blocking nature of protected objects, the locking mechanism of protected procedures and the entry queue depth of one all require new design methods to be developed when implementing multi-tasking, shared data, real-time systems.

The Aonix ObjectAda RAVEN product implements the Ceiling_Locking policy as a hard restriction (not optional). It should be noted that the Ravenscar Profile is silent on this issue. Using protected objects without this restriction can lead to behaviour that results in the generation of exceptions. In [9] it states that the rules regarding protected objects ensure that several tasks cannot be executing subprograms read and write simultaneously, however it says nothing about multiple write calls preempting each other via tasks of differing priorities. This could result in preemptive blocking within the protected object, which the Ravenscar Profile calls for a Program_Error to be raised if the Ceiling_Locking policy was not in place.

The solution of course lies within the use of both Ceiling_Locking and the specification of protected object priorities. In [2] it states that:

> Deadlocks can be prevented by assigning a priority to each protected object that is just greater than the highest priority of all its calling processes, and the use of Priority Ceiling Emulation within the run time via the locking policy Ceiling Locking, whereby a process using a protected object has its priority raised dynamically to the priority of the object. This method also allows a tight time bound for the priority inversion problem, in which a process may be blocked when attempting access to a protected object as a result of access to the same object by lower-priority processes.

It is our contention that the Ravenscar Profile should specify the use the Ceiling_Locking policy .

3.2 Exception Handling

Other papers have been published, notably [3] and [4], which have correctly asserted that the Ravenscar Profile has avoided defining how the tasking Profile handles exceptions. Steve Michell aptly points out that "exceptions are a language feature of Ada that cannot be avoided". Both the above mentioned papers discuss how exceptions can be added to the Ravenscar Profile, it is believed that a profile for exception handling should be defined at a system/language level rather than simply in the scope of the tasking profile.

Since the Ravenscar Profile did not specify how to handle exceptions, vendors such as Aonix were free to implement mechanisms that made sense to them. This in itself is a problem, as multiple vendors will ultimately develop different handling methods diluting the meaning of the Ravenscar Profile. However, the approach of the Aonix ObjectAda RAVEN product matched our needs and is an example of a system wide implementation.

As part of the kernel, Aonix has made visible several routines, which handle various conditions:

System.raven_instrumentation.handler_storage.adb
System.raven_instrumentation.sleep_processor.bdy
System.raven_instrumentation.task_storage.adb
System.raven_instrumentation.exception_handler.bdy
System.raven_instrumentation.task_controller.bdy

The exception_handler and task_controller routines are the ones primarily used for handling exceptions and dealing with the task termination event.

The procedure Exception_Handler is called by the kernel when an exception is raised. It is called in the thread of the task that generated the exception and the identity of this task is passed in as a parameter. This implementation allows for debuggers to collect data related to the fault and allows developers to implement routines which can allow for graceful system shutdown, or graceful system degradation, or whatever the specifications allow for after the detection of an exception.

The procedure Interrupt_Exception_Handler is called when an interrupt handler generates an exception. The identity is also passed in for use in a similar fashion as above.

The procedure Task_Termination is called by the kernel immediately before a task is permanently suspended. This routine allows developers with another means of performing system level remedial action, should they so desire.

The Raven Run-Time also internally configures handlers for several processor traps, which in turn are remapped to known exceptions. These exceptions are then handled via the Exception_Handler routine.

This implementation is a good example of a system wide exception handler implementation. The Aonix implementation allows for the user to insert some level of intervention before the task terminates (for example writing out to a software fault log). The determinism in jumping to a single known location for handling also makes this an acceptable solution. This methodology was successfully layered on top of another Ada95 kernel (one single handler). However, it is not clear that this model will always be the one of choice in developing real time systems because the model makes it difficult to do local recovery or provide locally degraded service. Our contention is that since the Ravenscar profile remains silent on the issue of exception handling that all vendor implementations in this regard should be implemented as optional features.

3.3 Dispatching

Dispatching results from the use of tagged type hierarchies and their realizations. Designing using abstract classes allows for the development of flexible code with respect to being able to call overloaded routines of the subclasses and having the runtime determine for whom the call was for, based on the class type. Designers should be aware that the overhead introduced in making dispatching calls might be significantly higher than making the call directly. The search through the dispatching table may not be the most efficient way of branching off to the next routine.

Our project has done some experimentation with dispatching and found that the Aonix compiler uses a table driven technology, which reduces the overhead, associated with the call, however we did notice on certain occasions where a table traversal seemed to be taking place as opposed to a direct call.

The use of dispatching may also introduce another unwanted side effect. One of the common requirements of critical real time software is that no dead code be left in the executable. A common toolset feature is the capability for removing unused code from the executable image. It was found, using the Aonix ObjectAda toolset, that the toolset radio button “Removed Unused Code” removed the unused dispatched code (the abstract realisations). It seems that when the controlling operands are dynamically determined (variables of a class wide type) and the tag checking is to be performed at run time somehow the toolset is unable to determine that code for the realization should remain in the system. It was subsequently discovered that since dispatching is not a recommended real-time systems programming design method, the Aonix compiler seems to apply this unadvertised prejudice. Thus when using the “Remove Unused Code” option, the dispatching code is removed. One can achieve a similar effect by using the Aonix soft restriction No_Dispatch which will generate compilation errors when dispatching code is encountered at the source code level.

4 Ada95 Ravenscar Design Guidelines

Designing with Ada95 and the Ravenscar Profile is still a relatively new field. The Ravenscar Profile itself is still maturing as more and more application designers become aware of it. It is also a fairly light specification, aimed at reducing the tasking complexity of the Ada language. For the most part this is completely acceptable, and it is acknowledged that over-specification is not always desirable. However, it would be helpful in the domain of real-time software development to produce more extensive examples and guidelines for developers to draw upon when designing to a certain specification. To this end, we present our solution to two issues within this domain, specifically the task start-up condition at elaboration that exists in Ada95 and the desire for best effort timing analysis and correction within a cyclic tasking system.

4.1 Task Start-up

At program start-up there exists a situation that applications developers have been forced to deal with for some time. The condition occurs due to the fact that in Ada tasks start immediately upon elaboration. In real time systems the requirement exists to understand the operation of and control the determinism of the software. Designers generally do not want their tasks running uncontrolled at start-up and typically must design some method to avert this situation.

Using the constructs available within the Aonix ObjectAda RAVEN, the start-up of the tasks can be controlled by requiring all tasks to issue an Ada.Synchronous_Task_Control.Suspend_Until_True call on a package level defined suspension object. All packages are required to implement:

```
procedure Run_Task( Run_State : in Boolean ) is
begin
        if Run_State then
                Ada.Synchronous_Task_Control.Set_True( SO );
        else
                Ada.Synchronous_Task_Control.Set_False( SO );
        end if;
end Run_Task;
```

The task loop would also have the Ada.Synchronous_Task_Control.Suspend_Until_True statement as it's first statement to allow for the task to be stopped during execution. The package containing the task would be required to implement body elaboration code that would set the suspension object by calling Ada.Synchronous_Task_Control.Set_False.

This method was chosen because it allows designers to select when to start their tasks, which has the added benefit of having a deterministic start-up from the mainline. Control and order are not the only issues with respect to starting tasks; initialization is also a key concern. Sometimes mainline initialization is desired before starting tasks, which is difficult to achieve under elaboration. This approach differs from other proposed solutions, such as a pragma, which would defer task start-up from occurring during elaboration and start them once the mainline starts. The problem with using the pragma in this manner is that is has not solved the issues of initialization or order, but has simply moved the problem.

4.2 Task Timing

Developing any system, including safety critical real-time systems, typically requires a series of timed tasks to be performed. All the elements for performing this functionality exist within the language and the Ravenscar Profile. What invariably occurs is that each system independently develops paradigms for calculating the timing for their system. The standard approach is to create tasks with the following Profile:

```
task body Cyclic is
begin
        loop
                delay until Next_Period;
                -- Task Execution (Te)
                Next_Period  := Next_Period + Period;

        end loop;
end Cyclic;
```

In this task, Period is set to a fixed time, Next_Period is usually set to an initial clock value therefore the task will be delaying at fixed Period intervals. This method is valid only under certain conditions, namely that the Te is not greater than the period, in which case the next delay until Period will be time in the past and the delay will not take place. This results in the erroneous execution of having the task spin (continually executing without delay) and starving out the lower priority tasks. Should the Te be some fraction of the period the task may execute in a desirable fashion, however if the Te rises in proportion to the Period the possibility of starvation of lower priority tasks rises as well.

Guidelines in the development of utilities for developing more reliable task timing structures should be part of a Profile relating to tasking. The following examples are those constructs developed using the Aonix ObjectAda Ravenscar development system.

These routines were added to an application Time package as an additional layer for use in conjunction with Ada.Real_Time and Ada.Real_Time.Raven:

```
 -- Description :  This function returns a the current 'time' at the moment
 --                of execution of this routine. (See Also Time_DIff)

function Time_Mark return Marker_Type is
  M : Marker_Type;
begin
  M.Time := Ada.Real_Time.Clock;
  return M;
end Time_Mark;

-- Description : This function calculates the time difference of the given
--               time parameter Time_Mark and the current 'time' and then
--               returns this value as an integer value of milliseconds.

function Time_Diff (Time_Mark : in Marker_Type) return Integer is

  Span : RT.Time_Span := RT.Clock - Time_Mark.Time;

begin
  -- Convert Span to duration then duration to Millisec (x 1000)
  return RTR.To_Microseconds (Span) / 1000;
end Time_Diff;

-- Description : Time value returned is Ada.Real_Time.Clock + value
--               Value must be expressed in milliseconds.
-- Returns a time value to be used with a "delay until" call.

function Time_Delay( Value : in Integer ) return Ada.Real_Time.Time is   -- input as milliseconds
begin
  return RT.Clock + RT.Milliseconds( Value );
end Time_Delay;
```

The use of these routines are shown in the following sample cyclic task:

```
task body Task_X is

   -- Local Variables
   Td  : Integer;
   T1: Time.Marker_Type;
   Te: Integer := 0;

begin

  loop

     T1 := Time.Time_Mark;

     -- Task Execution (Te)

     -- Calculate Execution Time
     Te := Time.Time_Diff( T1 );

     -- Calculate Desired Delay Time
     Td := Period - Te;

     -- Filter
     if Td < ( Period/Tuning_Divisor) then
        Td := ( Period/Tuning_Divisor );
     end if;

     delay_until Time.Time_Delay( Td );

  end loop;

end Task_X;
```

This particular philosophy provides several benefits to designers. Firstly the execution time (Te) can be calculated and then recorded, which helps designers understand their systems performance. Secondly the delay time required in order to meet the cyclic requirement defined by the period can be calculated (Td). We know that if Te > Period then the task will spin, starving lower priority tasks. What we also know is that if Td is < a certain proportion of the desired period then we may potentially starve out lower priority tasks. This implementation provides designers with the capability of tuning their system by adjusting the values of the Tuning_Divisor to prevent starvation from occurring.

Some effects arise when using this methodology. The filtering method (assigning a fixed value to Td) may result in Td + Te > Period, resulting in some system wide execution drift. Also, the delay until of Clock + Td may complete at the Period time interval, but the task may be a lower priority task resulting in pre-emptive blocking.

It is recognised that this methodology represents a best effort philosophy of software design, however this introduces increased code maintainability and safety out side of the safety critical paradigm envelope. It also provides better software design flexibility when developing Ravenscar software. From our experience, these techniques can be used to help isolate design errors during initial system development, errors with modifications during software maintenance phases, and can help provide designers with realistic values to be used with Rate Monotonic Analysis tools.

5 Conclusion

The Aonix ObjectAda RAVEN product is one of the first Ada95 development systems to have implemented the Ravenscar Profile and claim compliance. In many ways it has achieved it's goal successfully. The caveat is of course that the Ravenscar Profile should provide a mechanism for compiler vendors to indicate that they have included features above and beyond the standard. The pragma Ravenscar should be reserved as an indication that a set of strict features has been implemented. The standard should also allow for the creation of a vendor specific pragma *VendorX*_Ravenscar to indicate the superset of restrictions that have been included.

The Ravenscar Profile is a document summarising desirable features necessary to support the implementation of a standard for real-time systems software development. As such, there have been many companion documents and papers published with the intent of providing guidelines and interpretations on the use of language elements when developing software to meet the Ravenscar Profile. One of the purposes of this paper is to add to this knowledge base by providing further implementation examples in the domain of task start-up and cyclic task definitions.

In conclusion, the Ravenscar Profile as well as the Aonix ObjectAda RAVEN have achieved the goal of creating guidelines/products that software designers can use for the development of best effort software. Our initial expectations with respect to both have also been satisfactorily met. Participation in the debate over which areas of software development are appropriate for the Ravenscar Profile and/or how to achieve standards for the development of safety critical software are shared goals of CMC Electronics. It is felt that neither the Ravenscar Profile or the Aonix toolset meet the strict requirements defined within the discipline of safety critical real time software development and further refinement of both the Ravenscar Profile and the associated vendor products is still required if this is the goal.

6 References

[1] Burns, Alan, *The Ravenscar Profile*

[2] Aonix Corp., *ObjectAda Real-Time WindowsNT Intel/RAVEN Cross-Development Guide, UD/REF/A1920-05772/002 Aug 00*

[3] Michell, Stephen, *Position Paper: Completing the Ravenscar Profile*

[4] Ruiz, Jose, de la Peunte, Juan, Zamorano, Juan, Fernandez-Marina, Ramon, *Exception Support for the Ravenscar Profile, Ada-Letters, XXI, 3, September 2001*

[5] Lundqvist, Kristina, Asplund, Lars, Bjorkman, Mats, *A Run-Time System for Safety Critical Complex Systems*

[6] Dobbing, Brian, Humphries, David, *Ada Tasking for High-Integrity Systems*

[7] Romanski, George, *Safety Critical Software using Ada*

[8] Wellings, Andy, *Status and Future of the Ravenscar Profile Session Summary*

[9] Barnes, John, *Programming in Ada95*

[10] Kleidermacher, David and Griglock, Mark, *Safety-Critical Operating Systems*, Embedded Systems Programming, September 2001, Volume 14, Number 10.

[11] Wichmann, B A, *High Integrity Ada*